Rapport de projet – 1ère année

Heuristiques pour le voyageur de commerce

RAYNAUD Fabien

Responsable : P. MAHEY

Année 1999-2000

Sommaire

Introduction

Le problème du voyageur de commerce consiste à trouver un parcours de longueur minimum que doit emprunter un voyageur pour visiter une et une seule fois chaque ville s'il démarre de la ville de son domicile et y revient en fin de parcours. Ce problème est équivalent à la recherche d'un cycle hamiltonien minimal dans un graphe complet pondéré.

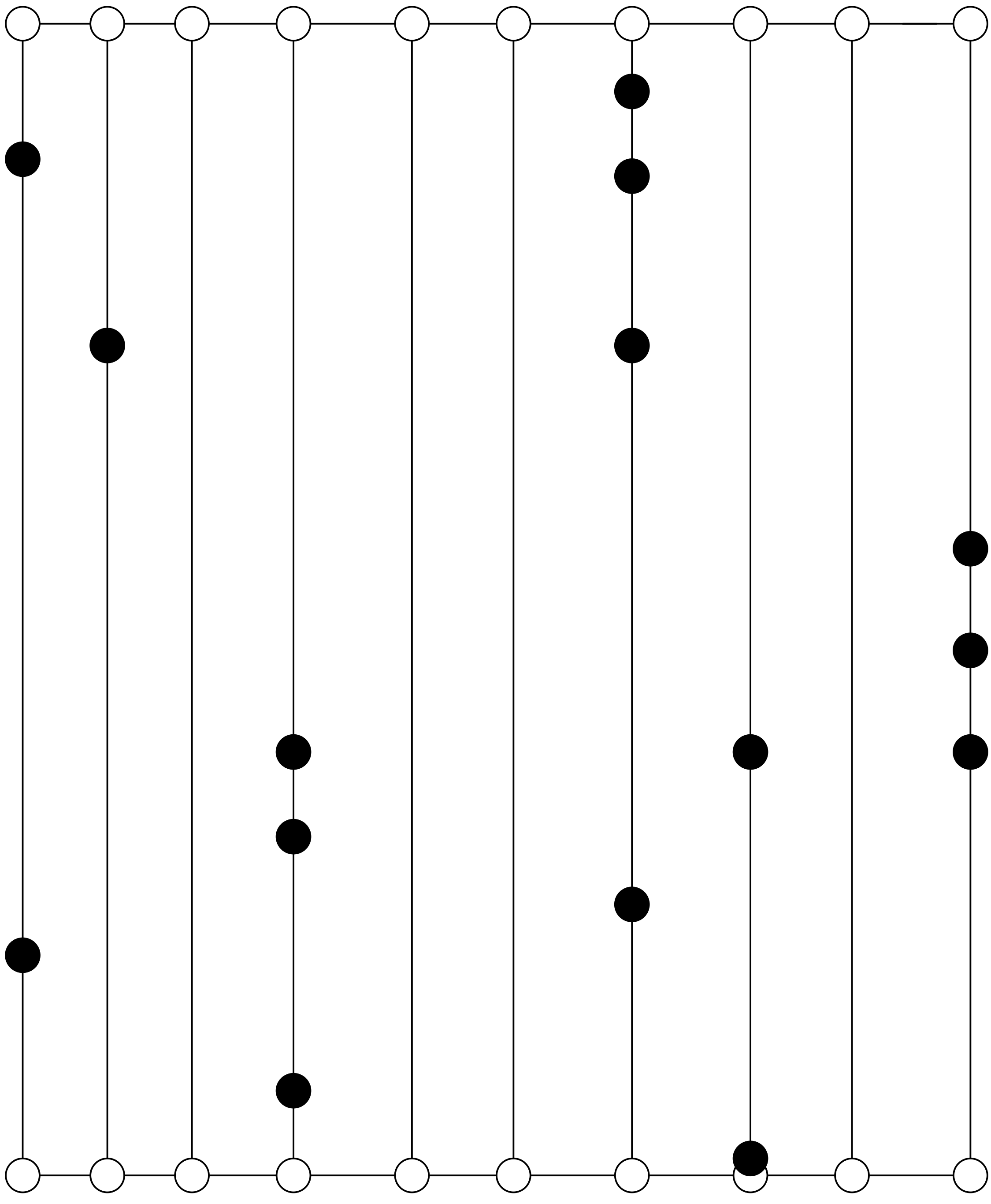
Ce problème est connu pour sa grande difficulté. Il est l'un des problèmes NP-difficiles les plus étudiés. La simplicité de son énoncé et les nombreuses applications qu'il présente en ont fait un problème très connu.

Le but de ce projet consiste à traduire les algorithmes classiques du Voyageur de Commerce en C, et à les tester sur différents exemples.

Application du Voyageur de Commerce

Le problème du voyageur de commerce peut s'appliquer à l'exemple suivant :

> Un entrepôt se présente sous la forme suivante : 2 allées horizontales et un certain nombre de travées verticales :



Un petit robot, basé au sommet (1) ,doit aller chercher un ensemble de colis situés sur les travées. La distance entre deux travées est de 2 et la position d’un colis est repérée par sa distance à allée horizontale la plus proche ou au colis voisin. Sur le dessin, les lieux des paquets sont les points noirs (que nous supposons numérotés de haut en bas et de gauche à droite).

Le problème posé se formule de la manière suivante : étant donnée une commande (les points noirs), trouver le plus petit trajet partant du sommet (1) et y retournant permettant d’aller chercher tous les paquets.

On en déduit alors la matrice des distances suivante (matrice carrée symétrique) :

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
1	0	2	12	6	15	16	20	13	14	16	23	23	29	25	26	27
2	2	0	10	8	17	18	18	15	16	18	25	25	27	27	28	29
3	12	10	0	14	15	14	10	25	26	26	19	23	17	29	28	27
4	6	8	14	0	17	18	16	15	16	18	25	26	23	27	28	29
5	15	17	15	17	0	1	5	16	17	19	16	20	14	26	25	24
6	16	18	14	18	1	0	4	17	18	20	15	19	13	25	24	23
7	20	18	10	16	5	4	0	21	20	18	11	15	9	21	20	19
8	13	15	25	15	16	17	21	0	1	3	10	12	16	14	15	16
9	14	16	26	16	17	18	20	1	0	2	9	13	15	15	16	17
10	16	18	26	18	19	20	18	3	2	0	7	15	13	17	18	19
11	23	25	19	25	16	15	11	10	9	7	0	12	6	18	17	16
12	23	25	23	26	20	19	15	12	13	15	12	0	6	18	17	16
13	29	27	17	23	14	13	9	16	15	13	6	6	0	12	11	10
14	25	27	29	27	26	25	21	14	15	17	18	18	12	0	1	2
15	26	28	28	28	25	24	20	15	16	18	17	17	11	1	0	1
16	27	29	27	29	24	23	19	16	17	19	16	16	10	2	1	0

Heuristiques classiques du Voyageur de Commerce

- Phase 1 :
  - « Au-Hasard »
  - « Glouton »
  - « Plus proche voisin »
  - « Insertion de moindre coût »
  - « Insertion la plus chère »
  - « Insertion la plus proche »

- Phase 2 : Amélioration locale
  - « 2-Opt »
  - « 3-Opt »
  - « Petit 3-Opt »

Lexique des notations

NBSOMMETS : Variable globale définissant le nombre de sommets du problème

Parcours : Tableau a 1 dimension, de taille NBSOMMETS+1. Il contient l'ordre dans lequel le parcours se fait. Le premier et le dernier élément du tableau sont identiques, pour mettre en évidence le cycle hamiltonien.

T : Tableau a 2 dimensions, de taille NBSOMMETS x NBSOMMETS. Il représente la matrice des distances. La distance entre le sommet i et le sommet j est donnée par T[i-1][j-1].

visite : Tableau a 1 dimension, de taille NBSOMMETS+1. Il indique si un sommet a été visité. visite[i]=1 : Le sommet n°i a été visité

Fonctionnement des programmes

Chaque algorithme est codé en C. Chaque fichier .C fait appel à un fichier *dist.h* (contenant le nombre de sommets et la matrice des distances) et à un module *distpar.c* (permettant de calculer la distance d'un parcours)

Pour tester chaque algorithme, on considérera le problème suivant :

Nombre de sommets : **6**

Matrice des distances :

	1	**2**	**3**	**4**	**5**	**6**
1	0	5	8	4	3	2
2	5	0	4	2	1	3
3	8	4	0	7	5	4
4	4	2	7	0	9	8
5	3	1	5	9	0	4
6	2	3	4	8	4	0

- <u>Fichier *dist.h* :</u>

```
/************************* dist.h **************************/
/*                                                           */
/*  Contient le nombre de sommets et la matrice des          */
/*  distances                                                */
/*************************************************************/


#define NBSOMMETS 6


int T[NBSOMMETS][NBSOMMETS] = {0,5,8,4,3,2,
                               5,0,4,2,1,3,
                               8,4,0,7,5,4,
                               4,2,7,0,9,8,
                               3,1,5,9,0,4,
                               2,3,4,8,4,0};
```

```
/******* Exemple du robot ********/

/*

#define NBSOMMETS 16

int T[NBSOMMETS][NBSOMMETS] =
{00,02,12,06,15,16,20,13,14,16,23,23,29,25,26,27,
 02,00,10, 8,17,18,18,15,16,18,25,25,27,27,28,29,
 12,10,00,14,15,14,10,25,26,26,19,23,17,29,28,27,
 06, 8,14,00,17,18,16,15,16,18,25,26,23,27,28,29,
 15,17,15,17,00,01,05,16,17,19,16,20,14,26,25,24,
 16,18,14,18,01,00,04,17,18,20,15,19,13,25,24,23,
 20,18,10,16,05,04,00,21,20,18,11,15, 9,21,20,19,
 13,15,25,15,16,17,21,00,01,03,10,12,16,14,15,16,
 14,16,26,16,17,18,20,01,00,02, 9,13,15,15,16,17,
 16,18,26,18,19,20,18,03,02,00,07,15,13,17,18,19,
 23,25,29,25,16,15,11,10, 9,07,00,12,06,18,17,16,
 23,25,23,26,20,19,15,12,13,15,12,00,06,18,17,16,
 29,27,17,23,14,13, 9,16,15,13,06,06,00,12,11,10,
 25,27,29,27,26,25,21,14,15,17,18,18,12,00,01,02,
 26,28,28,28,25,24,20,15,16,18,17,17,11,01,00,01,
 27,29,27,29,24,23,19,16,17,19,16,16,10,02,01,00};

*/
```

- <u>Fichier *distpar.c* :</u>

```
/*********************** distpar.c ************************/
/*                                                         */
/*  Contient la procédure  de calcul d'un parcours         */
/**********************************************************/

int distance_parcours( int [] );

int distance_parcours( int Parcours[] )
{
 int dist=0;
 int i;

 for ( i=0 ; i<NBSOMMETS ; i++)
        dist+=T[Parcours[i]-1][Parcours[i+1]-1];

 return dist;
}
```

Algorithme exhaustif

Principe :

Cet algorithme construit tous les chemins possibles et calcule leurs longueurs. C'est actuellement le seul algorithme capable de déterminer le plus court chemin passant une et une seule fois par tous les sommets.

Avec n villes, il y a $(n-1)!/2$ possibilités.

Dans l'exemple du robot, il y a 16 sommets, soit 653 837 184 000 possibilités.

Cette méthode est donc inutilisable, sauf si le nombre de villes est très petit.

Algorithme « Au Hasard »

Principe :

Cet algorithme consiste à permuter un sommet i avec un sommet j, choisi au hasard. Il génère une permutation Per de 1,…,n avec la loi uniforme sur l'ensemble des permutations de n éléments.

Algorithme général:

```
POUR i = n à 2 pas –1 FAIRE
        Echanger Per(i) avec Per(random(1,i))
FAIT
```

Algorithme de principe :

```
POUR i=1 à NBSOMMETS FAIRE              [Initialisation du Parcours initial]
        Parcours[i-1] = i ;
FAIT
Parcours[NBSOMMETS] = 1 ;

dist = distance_parcours(Parcours) ;    [Calcul de la distance du Parcours]
distprec = dist ;

Afficher(Parcours) ;

POUR i=NBSOMMETS-1 à 2 FAIRE
        j=hasard(1,i-1) ;               [hasard(l,m) : renvoie un entier au hasard
                                         compris entre l et m]
        temp=Parcours[i] ;
        Parcours[i]=Parcours[j] ;       [Permutation des sommets i et j]
        Parcours[j]=temp ;

        dist=distance_parcours(Parcours) ;

        SI (dist < distprec) ALORS
                Afficher(Parcours) ;
                distprec=dist ;
        FSI
FAIT
```

Code source :

```
/************************ hasard.c ************************/
/*                                                        */
/*                  Algorithme "Au-Hasard"                */
/**********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "dist.h"

extern int distance_parcours( int [] );

void main()
{
 int dist,          /* Distance calculée */
     distprec;      /* Distance calculée précédemment */
 int i,
     j,
     k,
     temp;

 int Parcours[NBSOMMETS+1];   /* Parcours */
 /* On part du sommet n°1 et on arrive au sommet n°1 */

 time_t t;

 /* Initialisation du générateur aléatoire */
 srand( (unsigned) time(&t) );

 /* Initialisation du Parcours */
 for ( i=1 ; i<= NBSOMMETS ; i++)
     Parcours[i-1]=i;
 Parcours[NBSOMMETS]=1;

 dist = distance_parcours(Parcours);
 printf("\nParcours : ");
 for (i=0 ; i<=NBSOMMETS ; i++)
      printf("%d ",Parcours[i]);
 printf("\t==> Distance = %d",dist);

 distprec=dist;

 for ( i=NBSOMMETS-1 ; i>=2 ; i--)
  {
   j= 1 + rand()%(i-1);           /* j choisi au hasard */

   temp=Parcours[i];              /* Permutation  */
   Parcours[i]=Parcours[j];       /*  des sommets */
   Parcours[j]=temp;              /*    i et j    */

   dist = distance_parcours(Parcours);
```

```
   if (dist<distprec)
    {
     printf("\nParcours : ");
     for (k=0 ; k<=NBSOMMETS ; k++)
          printf("%d ",Parcours[k]);
     printf("\t==> Distance = %d",dist);
     distprec=dist;
    }
  }
 printf("\n");
}
```

Explication :

On choisit, tout d'abord, un parcours initial, et on calcule sa distance.

A chaque permutation effectuée, on calcule la nouvelle distance du parcours ainsi obtenu. Si cette distance est inférieure à celle calculée précédemment, on affiche le parcours et la distance associée.

Exemples :

```
$ hasard

Parcours : 1 2 3 4 5 6 1        ==> Distance = 31
Parcours : 1 5 6 4 2 3 1        ==> Distance = 29

$ hasard

Parcours : 1 2 3 4 5 6 1        ==> Distance = 31
Parcours : 1 6 3 4 5 2 1        ==> Distance = 28

$ hasard

Parcours : 1 2 3 4 5 6 1        ==> Distance = 31
Parcours : 1 2 3 6 5 4 1        ==> Distance = 30
Parcours : 1 5 3 6 2 4 1        ==> Distance = 21
Parcours : 1 6 3 5 2 4 1        ==> Distance = 18

$ hasard

Parcours : 1 2 3 4 5 6 1        ==> Distance = 31
Parcours : 1 6 2 3 4 5 1        ==> Distance = 28
```

Algorithme « Glouton »

Principe :

Cet algorithme consiste à insérer au fur et à mesure les arcs de plus faible coût, en faisant attention de ne pas créer de cycle qui ne contiennent pas tous les sommets, et de sommet de degré strictement supérieur à 2.

Algorithme général :

```
Retenues = □
TANT QUE |Retenues| < n FAIRE
        Soit e l'arête non examinée de plus faible poids.
        L'arête e est examinée.
        SI (V , Retenues U {e}) ne contient pas de sommets de degré 3 et de cycle de
           longueur < n ALORS
                Retenues = Retenues U {e}
        FSI
FAIT
```

Algorithme de principe :

Structure utilisée pour le classement des arêtes (liste chaînée) :

cellule->valeur : Valeur du coût de l'arête i-j
cellule->ind_i : Indice i
cellule->ind_j : Indice j
cellule->suiv : Pointeur vers la cellule suivante

Procédure classement_arcs()

```
[Mise en place du 1er élément]
tete->valeur=distance(1,2) ;
tete->ind_i=1 ;
tete->ind_j=2 ;

POUR i=0 à NBSOMMETS-1 FAIRE
        POUR j= (2 si i=0, i+1 sinon) à NBSOMMETS-1 FAIRE
                cour=tete ;
                TANT QUE (distance(i+1,j+1) > (cour->suiv)->valeur
                                ET cour->suiv & NIL) FAIRE
                        cour=cour->suiv ;
                FAIT

                SI (distance(i+1,j+1) ☛ (cour->suiv)->valeur
```

```
            OU distance(i+1,j+1) > cour->valeur)  ALORS
               cell=ALLOC() ;        [Allocation mémoire]

               suiv=cour->suiv ;

               cell->valeur=distance(i+1,j+1) ;
               cell->ind_i=i+1 ;
               cell->ind_j=j+1 ;
               cell->suiv=suiv ;
               cour->suiv=cell ;
         SINON |SI ( distance(i+1,j+1) ⦸ cour->valeur) ALORS
               |     cell=ALLOC() ;        [Allocation mémoire]
               |     cell->valeur=distance(i+1,j+1) ;
               |     cell->ind_i=i+1 ;
               |     cell->ind_j=j+1 ;
               |     cell->suiv=tete ;
               |     tete=cell ;
               |FSI
         FSI
      FAIT
   FAIT
```

<u>Code source :</u>

```
/************************* glouton.c ************************/
/*                                                          */
/*                  Algorithme "Glouton"                    */
/************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "dist.h"

typedef struct cellule
{
 int               valeur;
 int               ind_i;
 int               ind_j;
 struct cellule  * suiv;
} cellule_t;

       void classement_arcs ();
extern int  distance_parcours( int [] );

cellule_t  * tete;

void main()
{
```

```
 int Parcours[NBSOMMETS+1] = {0};

 cellule_t * cour;

 tete = (cellule_t *) malloc ( sizeof(cellule_t) );

 classement_arcs();
 cour=tete;

 printf("Arcs du parcours : \n");

 while (cour)
  {
   printf("%d %d\n",cour->ind_i,cour->ind_j);
   cour=cour->suiv;
  }

free(tete);

}

void classement_arcs ()
{
 int i,
     j;

 cellule_t * cour;

 /* Mise en place du 1er élément */
 tete->valeur = T[0][1];
 tete->ind_i  = 1;
 tete->ind_j  = 2;
 tete->suiv   = NULL;

 for ( i=0 ; i<NBSOMMETS ; i++ )
  {
   for ( j= (i==0)?2:i+1 ; j<NBSOMMETS ; j++)
    {
     cour=tete;
     while ( T[i][j] > (cour->suiv)->valeur && cour->suiv )
      {
       cour = cour->suiv;
      }
     if ( T[i][j] >= (cour->suiv)->valeur
                    || T[i][j] > cour->valeur )
      {
       cellule_t * cell,
                 * suiv;
       cell = (cellule_t *) malloc ( sizeof(cellule_t) );

       suiv=cour->suiv;

       cell->valeur = T[i][j];
       cell->ind_i  = i+1;
       cell->ind_j  = j+1;
```

```
        cell->suiv   = suiv;
        cour->suiv   = cell;
       }
      else if ( T[i][j] <= cour->valeur )
       {
        cellule_t * cell;
        cell = (cellule_t *) malloc ( sizeof(cellule_t) );

        cell->valeur = T[i][j];
        cell->ind_i  = i+1;
        cell->ind_j  = j+1;
        cell->suiv    = tete;
        tete=cell;
       }
      }
     }
}
```

Explication :

Le classements des arêtes selon leur coût s'effectue à l'aide d'une liste chaînée.

Le code ci-dessus permet d'afficher toutes les arêtes dans l'ordre croissant de leur coût. Il faut ensuite éliminer les arêtes créant des cycles de longueur strictement inférieure à n et des sommets de degré 3.

Exemple :

4-7
2-4
7-8
2-8 ! Ne doit pas être pris en compte car on crée le cycle {2,4,7,8,2}

Afin d'éliminer ces arêtes, on peut chercher le sommet attaché au sommet 2 (c'est à dire le sommet 4) et indiqué que l'on a « visité » le sommet 2. On recommence avec le sommet 4. Quand tous les sommets ont été visités, on regarde si le dernier visité correspond au sommet que l'on veut insérer (ici, le sommet 8). Si c'est le cas, on n'insère pas l'arête associée.

On obtient alors le code C suivant :

```
while (cour)
  {
   liste[ind]  = cour->ind_i;
   liste[ind+1]= cour->ind_j;
   ind+=2;
   sauv=cour->ind_i;
   for (i=0 ; i<ind ; i++)   /*   début - fin     */
                              /* début : i pair   */
                              /* fin   : i impair */
    {
     if ( liste[i]==sauv && !visite[liste[i+1]] && !(i%2) )
      {
       visite[liste[i+1]]=1;
       sauv=liste[i+1];
       i=0;
      }
     if ( liste[i]==sauv && !visite[liste[i-1]] && (i%2)  )
      {
       visite[liste[i+1]]=1;
       sauv=liste[i-1];
       i=0;
      }
    }

   if ( sauv != cour->ind_j  )
      printf("\t%d - %d\n",cour->ind_i,cour->ind_j);
   else ind-=2;
   cour=cour->suiv;
  }
```

Cependant, cette solution ne marche pas !

Algorithme « Plus proche voisin »

Principe :

Cet algorithme consiste, à partir d'un sommet i d'insérer à chaque itération le sommet j, plus proche voisin du sommet i.

Algorithme général :

```
Visiter un sommet quelconque
TANT QUE il existe un sommet non visité FAIRE
        Visiter un sommet non visité le plus proche du dernier visité
FAIT
Retourner au point de départ
```

Algorithme de principe :

```
ind=0 ;
cycle=0 ;                           [Booléen]
i=hasard(1,NBSOMMETS) ;             [hasard(l,m) : renvoie un entier au hasard
                                     compris entre l et m]
Parcours[ind]=i ;
visite[i]=1 ;

TANT QUE (cycle=0) FAIRE
        k=1 ;
        TANT QUE ( visite[k]=1) FAIRE    [On se place sur le 1er sommet non visité]
                k=k+1 ;
        FAIRE
        min=distance(i,k) ;         [Renvoie la distance entre les sommets i et k]
        tmp=k ;

        POUR j=k+1 à NBSOMMETS FAIRE
        [Recherche du plus proche voisin]
                SI (distance(i,j)<min ET visite[j]=0) ALORS
                        min=distance(i,j) ;
                        tmp=j ;
                FSI
        FAIT

        ind=ind+1 ;
        Parcours[ind]=tmp ;
        visite[tmp]=1 ;
```

SI (visite[i]=1 pour i=1..NBSOMMETS) ALORS
    cycle=1 ;
FSI

i=tmp ;   [Remise à jour du nouveau sommet trouvé]

FAIT

ind=ind+1 ;
Parcours[ind]=Parcours[0] ;

Afficher(Parcours) ;

<u>Code source :</u>

```
/*********************** voisin.c ***************************/
/*                                                          */
/*                 Algorithme "Plus proche voisin"          */
/************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "dist.h"

extern int distance_parcours ( int [] );

void main()
{
 int i,
     j,
     k,
     tmp;
 int cycle = 0;                      /* Booléen */

 int visite[NBSOMMETS+1]   = {0};  /*   visite[i]=1 :  */
                                     /* Sommet n°i visité*/

 int Parcours[NBSOMMETS+1] = {0};

 int ind=0;                          /* Indice du tableau */
                                     /*     Parcours      */

 int min;                            /* Minimum */

 time_t t;

 /* Initialisation du générateur aléatoire */
 srand( (unsigned) time(&t) );

 i = 1 + rand()%(NBSOMMETS);       /* Sommet initial choisi */
                                     /*      au hasard        */
```

```
 Parcours[ind]=i;
 visite[i]=1;

 while ( !cycle )
  {
   k=1;
   /* On se place sur le premier sommet non visité */
   while ( visite[k] )
      k++;
   min=T[i-1][k-1];
   tmp=k;

   for ( j=k+1 ; j<=NBSOMMETS ; j++)    /* Recherche du plus */
                                        /*   proche voisin   */
    {
     if ( T[i-1][j-1] < min && !visite[j] )
      {
       min = T[i-1][j-1];
       tmp=j;
      }
    }
   ind++;
   Parcours[ind]=tmp;
   visite[tmp]=1;

   for ( i=1 ; i<=NBSOMMETS ; i++ )
   /* Calcul du booléen cycle */
    {
     if ( visite[i] )
        cycle=1;
     else
      { cycle=0;
        break;
      }
    }

   i=tmp;
  }

 ind++;
 Parcours[ind]=Parcours[0];

 printf("\nParcours : ");
 for ( i=1 ; i<=NBSOMMETS+1 ; i++)
     printf("%d ",Parcours[i-1]);

 printf("\t==> Distance = %d\n",distance_parcours(Parcours));

}
```

Explication :

On choisit au hasard un sommet initial. A chaque fois qu'un sommet est inséré dans le parcours, on indique qu'il a été visité (à l'aide du tableau visite[]).

Tant que tous les sommets n'ont pas été visités (ce qui correspond à cycle=0), on recherche le plus proche voisin des sommets du cycle n'ayant pas été visité.

Lorsque tous les sommets ont été visités (cycle=1), on revient au point de départ.

Exemples :

$ voisin

Parcours : 1 6 2 5 3 4 1 ==> Distance = 22

$ voisin

Parcours : 4 2 5 1 6 3 4 ==> Distance = 19

$ voisin

Parcours : 2 5 1 6 3 4 2 ==> Distance = 19

$ voisin

Parcours : 5 2 4 1 6 3 5 ==> Distance = 18

Algorithme « Insertion de moindre coût »

Principe :

A partir d'un cycle arbitraire de 3 points, cet algorithme insère entre deux sommets i et j adjacents le sommet k vérifiant $\min_k (\min_{i,j} (D_{ik} + D_{kj} – D_{ij}))$ (1).

Algorithme général :

```
Soit C un cycle constitué d'un triangle (arbitraire)
TANT QUE il existe des sommets non sur C FAIRE
        Insérer le sommet k non sur C entre les deux sommets i et j adjacents sur C si
        k, i, j satisfont min k∉C (min i,j consécutifs sur C (Dik + Dkj – Dij))
        Soit C le nouveau cycle
FAIRE
```

Algorithme de principe :

```
[Triangle initial]
Parcours[0]=1 ;
Parcours[1]=2 ;
Parcours[2]=3 ;
Parcours[3]=1 ;
ind=3 ;          [Indice du tableau Parcours]
visite[1]=visite[2]=visite[3]=1 ;

TANT QUE ( Parcours[NBSOMMETS] & 1) FAIRE       [Le cyle complet n'est ]
                                                [      pas terminé      ]
        k=1 ;
        TANT QUE (visite[k]=1) FAIRE     [On se place sur le 1er sommet non visité]
                k=k+1 ;
        FAIT
        min=distance(Parcours[0],k)+distance(Parcours[1],k)-
              distance(Parcours[0],Parcours[1]) ;
        itmp=0 ;        [Variables de]
        jtmp=k ;        [ sauvegarde ]

        POUR i=1 à ind-1 FAIRE
                POUR j=k à NBSOMMETS FAIRE
                        SI (distance(Parcours[i],j)+distance(Parcours[i+1],j)-
                              distance(Parcours[i],Parcours[i+1]) < min
                              ET visite[j]=0) ALORS
```

```
min= distance(Parcours[i],j)+distance(Parcours[i+1],j)-
        distance(Parcours[i],Parcours[i+1]) ;
itmp=i ;
jtmp=j ;
            FSI
        FAIT
    FAIT

    [Insertion du nouveau sommet]
    POUR i=ind+1 à itmp+1 pas –1 FAIRE
        Parcours[i]=Parcours[i-1] ;
    FAIT
    Parcours[itmp+1]=jtmp ;

    visite[jtmp]=1 ;
    ind=ind+1 ;
FAIT

Afficher(Parcours) ;
```

Code source :

```
/*********************** insmcout.c ***********************/
/*                                                         */
/*         Algorithme "Insertion de moindre coût"          */
/***********************************************************/

#include <stdio.h>
#include "dist.h"

extern int distance_parcours( int [] );

void main()
{
 int i,
     j,
     k,
     itmp,
     jtmp;

 int visite[NBSOMMETS+1]   = {0};   /* visite[i]=1 : Sommet */
                                    /*      n°i visité      */

 int Parcours[NBSOMMETS+1] = {0};
 int ind=3;                   /* Indice du tableau Parcours */
 int min;                     /* Minimum */


 /* Triangle initial */
 Parcours[0]=1;
 Parcours[1]=2;
```

```
 Parcours[2]=3;
 Parcours[3]=1;

 visite[1]=visite[2]=visite[3]=1;

 while ( Parcours[NBSOMMETS] != 1 )
  {
   k=1;
   while ( visite[k] )
       k++;
   min=T[Parcours[0]-1][k-1]+T[Parcours[1]-1][k-1]-
       T[Parcours[0]-1][Parcours[1]-1];
   itmp=0;
   jtmp=k;

   for ( i=0 ; i<ind ; i++ )       /* Recherche du plus */
                                   /*   proche voisin   */
    {
     for ( j=k ; j<=NBSOMMETS ; j++)
      {
       if ( T[Parcours[i]-1][j-1]+T[Parcours[i+1]-1][j-1]-
            T[Parcours[i]-1][Parcours[i+1]-1]
            < min && !visite[j] )
        {
         min = T[Parcours[i]-1][j-1]+T[Parcours[i+1]-1][j-1]-
               T[Parcours[i]-1][Parcours[i+1]-1];
         itmp=i;
         jtmp=j;
        }
      }
    }

   /* Insertion du nouveau sommet */
   for ( i=ind+1 ; i>itmp ; i--)
     Parcours[i]=Parcours[i-1];
   Parcours[itmp+1]=jtmp;

   visite[jtmp]=1;
   ind++;
  }


 printf("\nParcours : ");
 for ( i=1 ; i<=NBSOMMETS+1 ; i++)
  {
   printf("%d ",Parcours[i-1]);
  }
 printf("\t==> Distance = %d\n",distance_parcours(Parcours));

}
```

Explication :

A partir du cycle initial 1,2,3,1, on détermine le sommet satisfaisant la condition (1). A chaque fois qu'un sommet est inséré, on indique qu'il a été visité (à l'aide du tableau visite[]). On continue ainsi, tant que le cycle complet n'a pas été effectué.

Exemples :

$ insmcout

Parcours : 1 5 2 4 3 6 1 ==> Distance = 19

Algorithme « Insertion la plus chère »

Principe :

A partir d'un cycle arbitraire de 3 points, cet algorithme insère entre deux sommets i et j adjacents le sommet k vérifiant $\max_k (\min_{i,j} (D_{ik} + D_{kj} - D_{ij}))$

Algorithme général :

Soit C un cycle constitué d'un triangle (arbitraire)
TANT QUE il existe des sommets non sur C FAIRE
    Insérer le sommet k non sur C entre les deux sommets i et j adjacents sur C si k, i, j satisfont $\max_{k \notin C} (\min_{i,j \text{ consécutifs sur } C} (D_{ik} + D_{kj} - D_{ij}))$
    Soit C le nouveau cycle
FAIRE

Algorithme « Insertion la plus proche »

<u>Principe :</u>

A partir d'un cycle arbitraire de 3 points, cet algorithme insère le sommet le plus proche d'un des sommets du cycle, après le sommet dont il est le plus proche.

<u>Algorithme :</u>

```
Soit C un cycle (arbitraire) constitué de 3 sommets
TANT QUE il existe des sommets non sur C FAIRE
        Soit x le sommet non sur C le plus proche d'un sommet de C
        Insérer x après le sommet dont il est le plus proche
        Soit C le nouveau cycle
FAIT
```

<u>Algorithme de principe :</u>

```
[Triangle initial]
Parcours[0]=1 ;
Parcours[1]=2 ;
Parcours[2]=3 ;
Parcours[3]=1 ;
ind=3 ;          [Indice du tableau Parcours]
visite[1]=visite[2]=visite[3]=1 ;

TANT QUE ( Parcours[NBSOMMETS] & 1) FAIRE        [Le cycle complet n'est ]
                                                 [     pas terminé      ]
        k=1 ;
        TANT QUE (visite[k]=1) FAIRE     [On se place sur le 1er sommet non visité]
                k=k+1 ;
        FAIT
        min=distance(Parcours[0],k) ;
        itmp=0 ;        [Variables de]
        jtmp=k ;        [ sauvegarde ]

        POUR i=0 à ind-1 FAIRE
                POUR j=k à NBSOMMETS FAIRE
                        SI (distance(Parcours[i],j) < min ET visite[j]=0) ALORS
                                min=distance(Parcours[i],j) ;
                                itmp=i ;
                                jtmp=j ;
                        FSI
```

FAIT

FAIT

[Insertion du nouveau sommet]

POUR i=ind+1 <u>à</u> itmp+1 <u>pas</u> –1 FAIRE

Parcours[i]=Parcours[i-1] ;

FAIT

Parcours[itmp+1]=jtmp ;

visite[jtmp]=1 ;

ind=ind+1 ;

FAIT

Afficher(Parcours) ;

<u>Code source :</u>

```
/*********************** insproch.c ************************/
/*                                                          */
/*          Algorithme "Insertion la plus proche"          */
/***********************************************************/

#include <stdio.h>
#include "dist.h"

extern int distance_parcours( int [] );

void main()
{
 int i,
     j,
     k,
     itmp,
     jtmp;
 int visite[NBSOMMETS+1]   = {0};   /* visite[i]=1 :Sommet */
                                   /*      n°i visité     */

 int Parcours[NBSOMMETS+1] = {0};

 int ind=3;                       /* Indice du tableau */
                                  /*      Parcours     */

 int min;                         /* Minimum */

 /* Initialisation du parcours */
 Parcours[0]=1;
 Parcours[1]=2;
 Parcours[2]=3;
 Parcours[3]=1;

 visite[1]=visite[2]=visite[3]=1;
```

```
 while ( Parcours[NBSOMMETS] != 1 )
  {
   k=1;
   while ( visite[k] )
      k++;
   min=T[Parcours[0]-1][k-1];
   itmp=0;
   jtmp=k;

   for ( i=0 ; i<ind ; i++ )       /* Recherche du plus  */
                                   /* proche voisin d'un */
                                   /*   sommet du cycle  */
    {
     for ( j=k ; j<=NBSOMMETS ; j++)
      {
       if ( T[Parcours[i]-1][j-1] < min && !visite[j] )
        {
         min = T[Parcours[i]-1][j-1];
         itmp=i;
         jtmp=j;
        }
      }
    }

   /* Insertion du sommet, par décalage */
   for ( i=ind+1 ; i>itmp ; i--)
       Parcours[i]=Parcours[i-1];
   Parcours[itmp+1]=jtmp;

   visite[jtmp]=1;
   ind++;
  }

 printf("\nParcours : ");
 for ( i=1 ; i<=NBSOMMETS+1 ; i++)
     printf("%d ",Parcours[i-1]);

 printf("\t==> Distance = %d\n",distance_parcours(Parcours));

}
```

Explication :

On crée le cycle initial contenant les 3 premiers sommets. A chaque fois qu'un sommet est inséré dans le cycle, on indique qu'il a été visité à l'aide du tableau visite[].

Tant que le cycle entier n'a pas été effectué, on recherche le sommet non visité qui est le proche d'un sommet du cycle. On insère ensuite ce sommet après le sommet dont il est le plus proche.

Exemple :

$ insproch

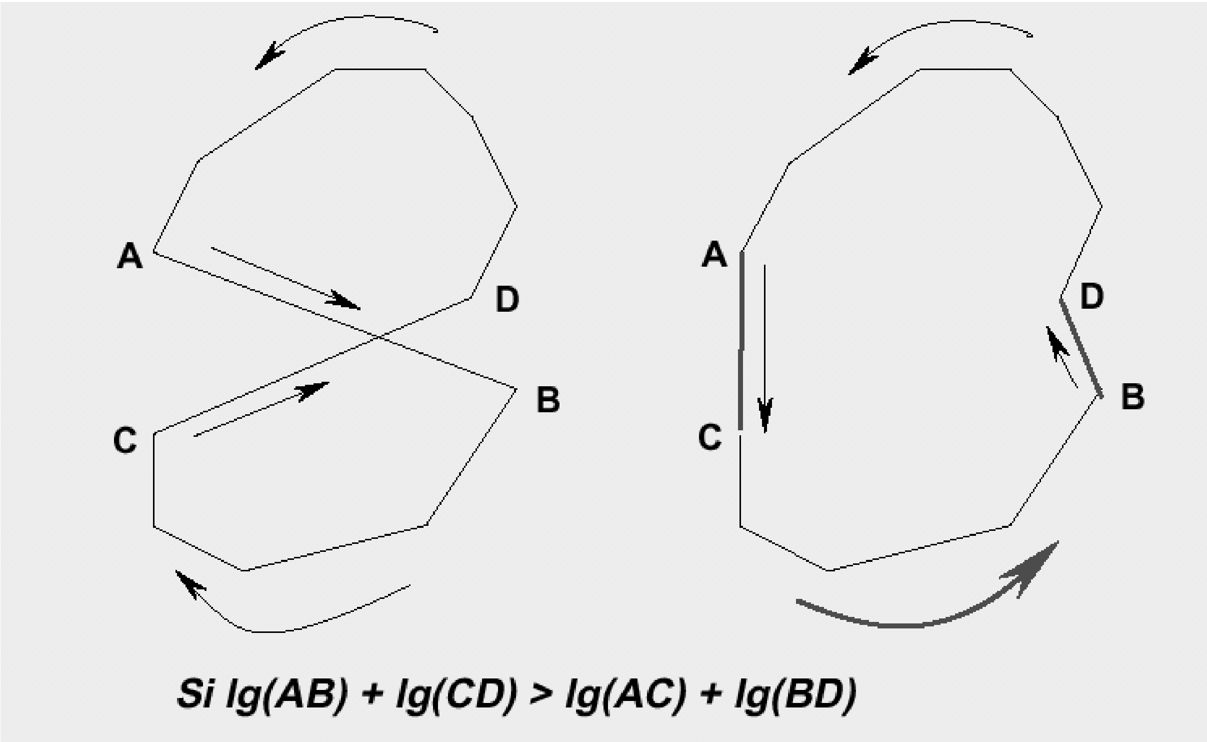
Parcours : 1 6 2 4 5 3 1   ==> Distance = 29

Algorithme « 2-Opt »

Principe :

A partir d'un cycle hamiltonien, cet algorithme retire 2 arêtes du cycle, et calcule un autre cycle en remplaçant ces arêtes par celles appropriées.

Exemple :



Algorithme général :

Soit C un cycle quelconque
Etablir la liste L des couples d'arêtes de C non adjacentes
TANT QUE L est non vide FAIRE
    Soit l un élément de L
    Retirer l de L
    Obtenir un cycle hamiltonien C' en retirant de C les deux arêtes de l et en les remplaçant par deux arêtes appropriées…
    SI C' est meilleur que C ALORS
        C=C'
    FSI
FAIT

Algorithme de principe :

```
Lire(Parcours) ;
P=Parcours ;

POUR i=0 à NBSOMMETS-1 FAIRE
    POUR j=i+2 à NBSOMMETS-1 FAIRE
        k=i+1 ;
        l=j ;
        TANT QUE (k<l) FAIRE
        [Inversion de l'ordre des sommets entre les indices i+1 et j]
            tmp=Parcours[k] ;
            Parcours[k]=Parcours[l] ;
            Parcours[l]=tmp ;
            k=k+1 ;
            l=l-1 ;
        FAIT
        SI ( distance_parcours(Parcours) > distance_parcours(P) ) ALORS
               Parcours=P ;
        SINON   P=Parcours ;
        FSI
    FAIT
FAIT

Afficher(Parcours) ;
```

Code source :

```
/**************************** 2opt.c ************************/
/*                                                          */
/*                    Algorithme "2-Opt"                    */
/************************************************************/

#include <stdio.h>
#include "dist.h"

extern int distance_parcours( int [] );

void main()
{
 int Parcours[NBSOMMETS+1];
 int P       [NBSOMMETS+1];
 int i,
     j,
     k,
     l,
```

```
    tmp;

 /* Lecture du Parcours au clavier, en affichant NBSOMMETS */
 printf("Indiquez le parcours (en n'oubliant pas de boucler
         sur le sommet initial)\n");
 printf("Nombre de sommets : %d\n",NBSOMMETS);
 for (i=0 ; i<=NBSOMMETS ; i++)
     scanf("%d",&Parcours[i]);
 for ( k=0 ; k<=NBSOMMETS ; k++)
     P[k]=Parcours[k];


 for ( i=0 ; i<NBSOMMETS ; i++ )
  {
   for ( j=i+2 ; j<NBSOMMETS ; j++ )
    {
     k=i+1;
     l=j;
     while ( k<l )
     /* Inversion de l’ordre des sommets entre i+1 et j*/
      {
       tmp=Parcours[k];
       Parcours[k]=Parcours[l];
       Parcours[l]=tmp;
       k++;
       l--;
      }

     if ( distance_parcours(Parcours) > distance_parcours(P) )
      {
       for ( k=0 ; k<=NBSOMMETS ; k++)
           Parcours[k]=P[k];
      }
     else
      {
       for ( k=0 ; k<=NBSOMMETS ; k++)
           P[k]=Parcours[k];
      }
    }
  }

 printf("\nParcours : ");
 for ( i=1 ; i<=NBSOMMETS+1 ; i++)
  {
   printf("%d ",Parcours[i-1]);
  }

 printf("\t==> Distance = %d\n",distance_parcours(Parcours));

}
```

Explication :

L'utilisateur entre le parcours initial au clavier. On recopie Parcours dans P. On parcourt la liste Parcours à l'aide de 2 pointeurs *i* et *j*. En faisant varier ces deux indices, on obtient la liste des arêtes non adjacentes (car *j* varie de *i+2* à NBSOMMETS-1). Le changement d'arêtes de l'algorithme « 2-Opt » consiste à inverser l'ordre des sommets dans la liste Parcours entre les indices *i+1* et *j* inclus (comme l'illustre le schéma vu précédemment). Si la distance de Parcours est supérieure à celle de P (on a effectué un Parcours plus long que précédemment), on recopie P dans Parcours, sinon on recopie Parcours dans P.

Exemples :

$ 2opt
Indiquez le parcours (en n' oubliant pas de boucler sur le sommet initial)
Nombre de sommets : 6
1 2 3 4 5 6 1

Parcours : 1 6 3 5 2 4 1   ==> Distance = 18

$ 2opt
Indiquez le parcours (en n' oubliant pas de boucler sur le sommet initial)
Nombre de sommets : 6
2 4 5 1 3 6 2

Parcours : 2 6 5 3 1 4 2   ==> Distance = 26

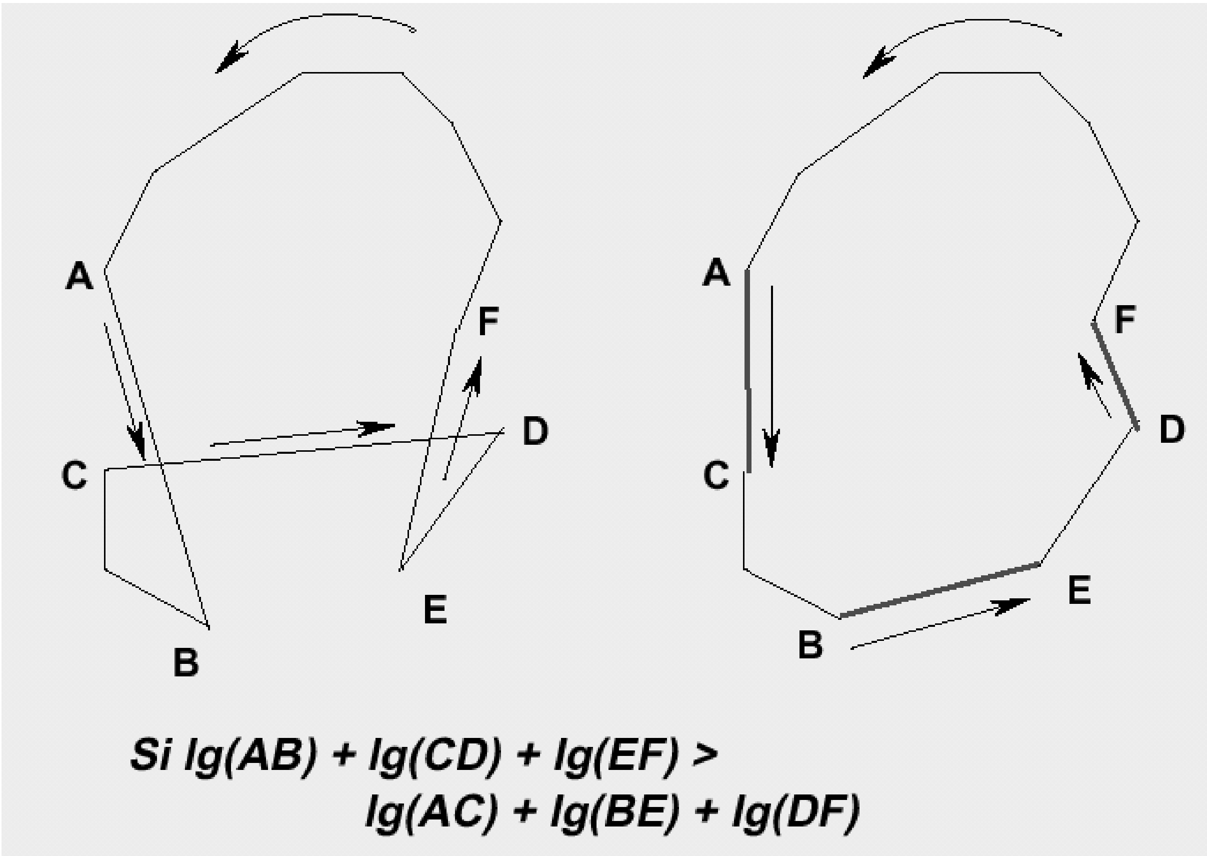
$ 2opt
Indiquez le parcours (en n' oubliant pas de boucler sur le sommet initial)
Nombre de sommets : 6
3 4 2 1 6 5 3

Parcours : 3 5 2 4 1 6 3   ==> Distance = 18

Algorithme « 3-Opt »

<u>Principe :</u>

A partir d'un cycle hamiltonien, cet algorithme retire 3 arêtes du cycle, et calcule un autre cycle en remplaçant ces arêtes par celles appropriées.



Remarques sur 2-Opt et 3-Opt :

Pour n sommets :

2-Opt : Nombre de voisinages = $n(n-3)/2$
3-Opt : Nombre de voisinages = $n(n-3)(n-2)$

En général, on part d'un chemin aléatoire (ou avec algorithme du plus proche voisin) et on applique une transformation tant que l'on peut.

En général, 3-Opt est plus efficace que 2-Opt, mais est plus coûteux en temps, ce qui fait qu'on utilise plus souvent 2-Opt.

Algorithme « Petit 3-Opt »

Algorithme général :

```
Soit C un cycle hamiltonien quelconque
V’=V
TANT QUE  V’ & □  FAIRE
        Soit x un sommet de V’
        V’=V’\{x}
        Obtenir un cycle hamiltonien C’ en supprimant de C les deux arêtes incidentes
        à x, en fermant la chaîne obtenue, puis en insérant x entre deux sommets
        consécutifs du cycle ainsi obtenu
        SI C’ est meilleur que C ALORS
                C=C’
                V’=V
        FSI
FAIT
```

Applications

<u>*Exemple du robot :*</u>

- Algorithme « Au-Hasard » :

  $ hasard

  Parcours : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 1 ==> Distance = 138
  Parcours : 1 2 3 4 5 6 7 8 9 10 11 12 13 16 15 14 1 ==> Distance = 134

On ne parvient pas à obtenir de meilleurs résultats car la numérotation initiale donne un « bon » parcours.

- Algorithme « Plus proche voisin » :

  $ voisin

  Parcours : 15 14 16 13 11 10 9 8 12 7 6 5 1 2 4 3 15 ==> Distance = 128

  $ voisin

  Parcours : 3 2 1 4 8 9 10 11 13 12 7 6 5 16 15 14 3 ==> Distance = 130

  $ voisin

  Parcours : 8 9 10 11 13 12 7 6 5 1 2 4 3 16 15 14 8 ==> Distance = 124

  $ voisin

  Parcours : 12 13 11 10 9 8 1 2 4 3 7 6 5 16 15 14 12 ==> Distance = 118

  $ voisin

  Parcours : 4 1 2 3 7 6 5 13 11 10 9 8 12 16 15 14 4 ==> Distance = 120

- Algorithme « Insertion de moindre coût » :

  $ insmcout

  Parcours : 1 2 3 7 6 5 14 15 16 12 13 11 10 9 8 4 1   ==> Distance = 114

- Algorithme « Insertion la plus proche » :

  $ insproch

  Parcours : 1 4 2 3 7 13 16 15 14 12 11 10 9 8 6 5 1   ==> Distance = 128

- Algorithme « 2-Opt » :

  $ 2opt

  Parcours : 1 16 14 15 8 9 10 11 12 13 7 6 5 3 2 4 1   ==> Distance = 126

On peut appliquer ces algorithmes sur d'autres exemples, en changeant la matrice des distances qui se trouve dans le fichier *dist.h*. En mémoire statique, la limitation du C pour les matrices est de l'ordre de 150x150.

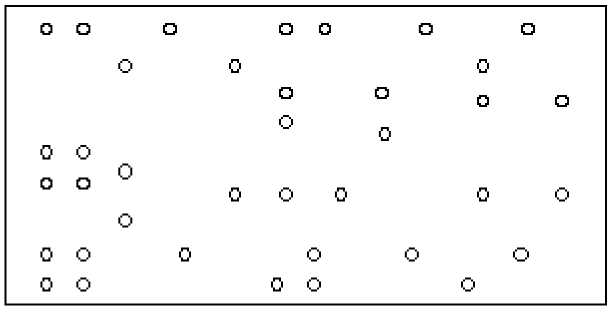
Pour plus de 150 sommets, il faut avoir recours à l'allocation de mémoire dynamique.

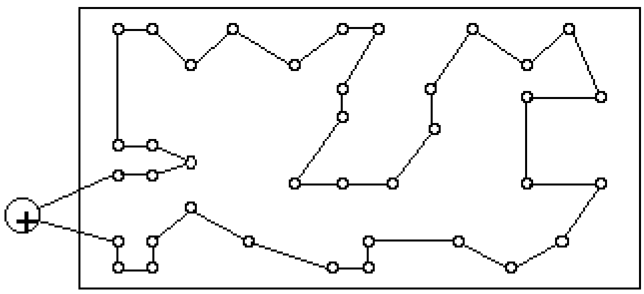
<u>Autres applications du Voyageur de Commerce :</u>

Le problème du voyageur de commerce présente de nombreuses applications dans différents domaines : télécommunications, logistique, électronique, etc…

Par exemple, en électronique, on peut imaginer le problème suivant :

Un robot doit relier électriquement un ensemble de points et revenir à sa position initiale, afin de traiter une nouvelle plaque.





Commentaires

Pour chaque problème, il faut créer la matrice des distances. Mais, on peut supposer qu'un programme donne les coordonnées des deux sommets *i* et *j*, et ainsi puisse calculer la distance *i-j*.

Pour faciliter le changement de matrice, on peut stocker la matrice dans un fichier, puis on lit ce fichier que l' on stocke dans un tableau a 2 dimensions. Ceci afin d' éviter un accès disque à chaque demande de distance.

Du fait de la même structure du parcours dans chaque algorithme (tableau à 1 dimension), cela peut permettre une communication entre les algorithmes de la Phase 1 et de la Phase 2.

Dans la pratique, on calcule un parcours à l'aide d'un algorithme de la Phase 1 puis ce parcours est amélioré avec un algorithme de la Phase 2 (phase d'amélioration locale).

Dans le cas de l'allocation statique de mémoire, le nombre de sommets est limité à environ 150. Pour remédier à cela, il faut utiliser la gestion dynamique de la mémoire.

Ainsi, la matrice *n* x *n* sera considéré comme un tableau a 1 dimension de taille n^2 et l'élément T[i][j] sera identifié par *(T + (i*n + j))

Exemple :

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define NBSOMMETS 500

void main()
{
 int  * T;
 int    i,
        j;
 time_t t;

 srand( (unsigned) time(&t) );

 T = (int *) malloc ( NBSOMMETS*NBSOMMETS * sizeof(int) );

 for (i=0;i<NBSOMMETS;i++)
  {
   for (j=0;j<NBSOMMETS;j++)
    {
     if (i==j) *(T+(i*NBSOMMETS+i))=0;
      *(T + (i*NBSOMMETS+j) ) = rand()%(1000);
      *(T + (j*NBSOMMETS+i) ) = *(T + (i*NBSOMMETS+j) ) ;
    }
  }


free(T);

}
```

Conclusion

Les heuristiques de la Phase 1 permettent de construire un parcours. Les algorithmes de la Phase 2 (appelée phase d'amélioration locale) permettent d'améliorer le parcours ainsi trouvé.

L'algorithme « Au-Hasard » peut paraître simpliste, mais il donne des résultats corrects. L'algorithme « Plus proche voisin » donne d'assez bon résultats, excepté pour certaines configurations de sommets.

Les meilleurs algorithmes de la Phase 1 sont :

1) Glouton
2) Insertion de moindre coût
3) Plus proche voisin

Ces résultats dépendent évidemment de la place des sommets.

L'algorithme « 3-Opt » est plus efficace que « 2-Opt ». Mais « 3-Opt » est plus coûteux en temps. On utilisera donc « 2-Opt » le plus souvent.

Ainsi, pour obtenir les meilleurs résultats, il faut combiner un algorithme de la Phase 1 et un autre de la Phase 2. Dans la pratique, on part d'un chemin (avec algorithme du plus proche voisin ou glouton ou hasard) et on applique une transformation tant que l'on peut.